Getting Started with RFNoC in UHD 4.0

Contents

AN-400 by Sugandha Gupta, Brent Stapleton, Wade Fife, and Michael Dickens

This guide describes how to get started with FPGA and Software development for RF Network-on-Chip (RFNoC?). It gives a brief introduction to RFNoC and explains the steps needed to generate, build, and use custom RFNoC images and introduces the process for creating and integrating new RFNoC IP blocks.

This guide is written for hardware and software engineers who want to use the RF Network-on-Chip (RFNoC?) architecture or want to develop intellectual property (IP) using the RFNoC architecture. For more details on the architecture please refer to the RFNoC Specification.

This guide assumes that you have some basic familiarity with USRPs, such as connecting them, configuring your network interfaces, etc., so that your USRP is ready for use. See the Getting Started Guide for your USRP if you are just getting started with USRPs.

The RFNoC code base is open source, including code that executes on the host, as well as code targeted to the USRP hardware (FPGA and microcontroller firmware). RFNoC is available under the open-source GNU Lesser General Public License (LGPL). For more information on our licensing policy, please contact info@ettus.com.

RFNoC is currently supported on the USRP X410 series, and all the Generation-3 USRPs in the X series (X3xx), E series (E3xx) and N series (N3xx). For details on the hardware, software, and process required to build custom USRP FPGA images that include RFNoC blocks, see the USRP Build Documentation in the UHD and USRP Manual.

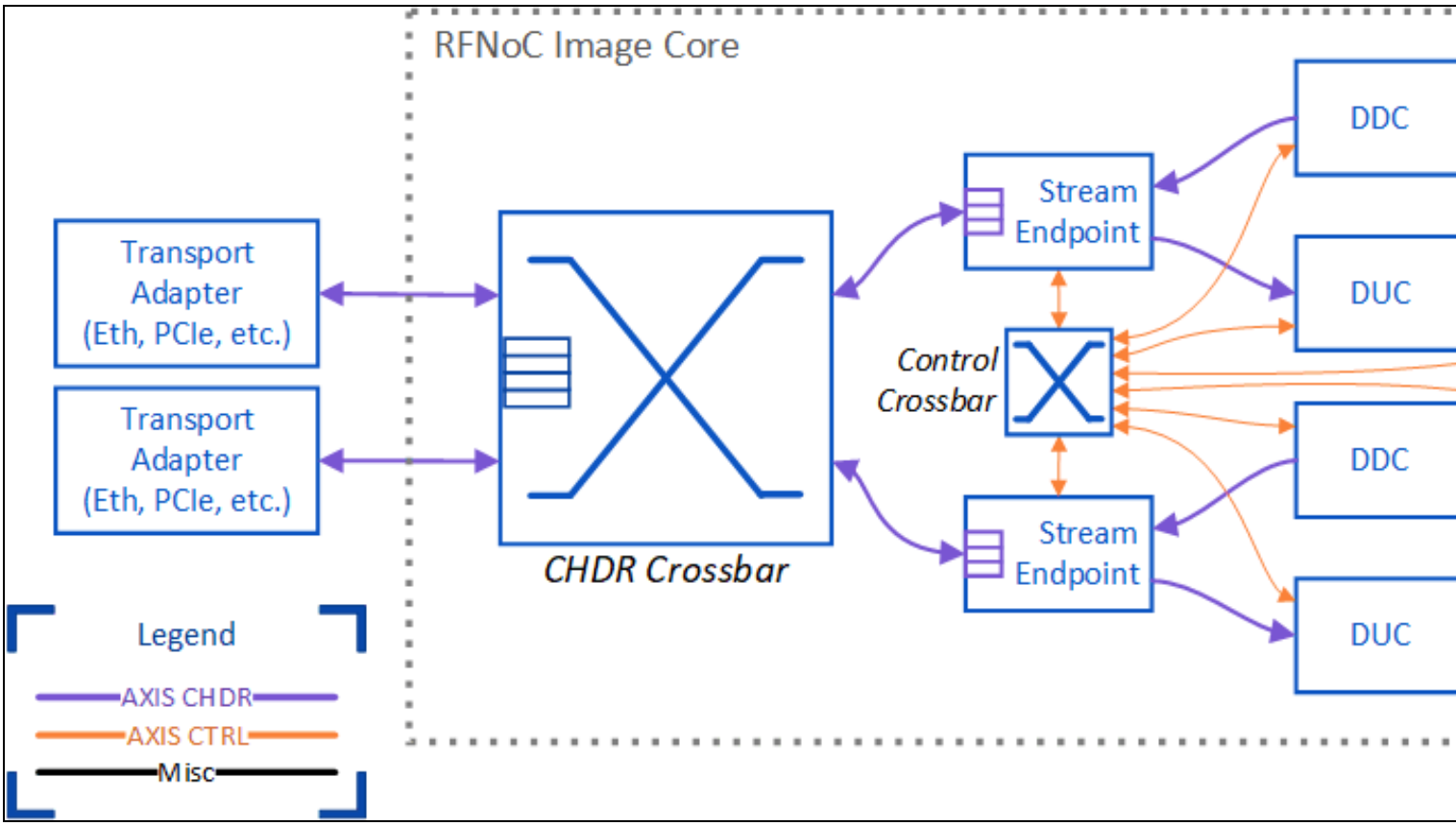
It is recommended that you learn how to build an FPGA image for your USRP and download it to the device before starting this guide if you have never done so before. This will ensure you have all the necessary software installed.

RFNoC? is a heterogeneous processing framework used to implement high-throughput DSP in the FPGA for Software Defined Radio (SDR) systems in an easy-to-use and flexible way. It provides all the infrastructure required to insert signal-processing IP into the FPGA logic and communicate with it through software. It also provides highly-optimized software and FPGA code to enable high-performance streaming to and from IP blocks on the USRP device.

The IP blocks in RFNoC are called *RFNoC blocks*. The RFNoC blocks wrap the IP and provide a custom interface to the RFNoC infrastructure through a tool-generated interface called the *NoC Shell*. Many standard blocks are included in UHD 4.0. These blocks enable typical operation of the USRP and allow RFNoC to connect to the different hardware components of the USRP. Several of the included blocks are described in the section Available RFNoC Blocks. You can mix the available blocks for your application, or develop your own custom RFNoC blocks with your own IP to open up new applications. The NoC Shell hides the complexity of RFNoC from your block, making it easy to plug your IP into the USRP.

RFNoC is used on all Generation-3 USRPs and is installed with UHD 4.0. USRPs ship with a default RFNoC image that can be used as is, or can be modified and/or customized to suite your application. So if you're using a modern USRP, it's running RFNoC under the hood even if you haven't customized anything.

The diagram below shows a simplified view of an RFNoC FPGA image that is similar to the default images available on USRPs.



Notice that an RFNoC FPGA is made up of several components. A description of each one is provided below.

- **RFNoC Image Core**
  This is the main block of the RFNoC framework and instantiates the components that make up RFNoC. The contents of the image core are described by an RFNoC image core YAML file, and the Verilog code that represents this block is automatically generated by the RFNoC Image Builder tool based on the YAML description. In other words, you provide a description of what you want to be included in RFNoC and the tools generate the code to implement that description.
- **Transport Adapter**
  The transport adapter is a component of the USRP that allows communication with an outside interface. These vary depending on the USRP model in use. For example, this could be a 10 Gbps Ethernet link for SFP+ port of the USRP, or a cabled PCIe interface.
- **CHDR Crossbar**
  The CHDR crossbar is a dynamic router for RFNoC traffic. This is a high-throughput crossbar designed for RF streaming applications. It is reconfigurable at run time via software and can be used to configure routes between stream endpoints and transport adapters. The number of ports available is configurable by changing the RFNoC image core YAML file. CHDR (Condensed Hierarchical Datagram for RFNoC) refers to the network protocol that is used for RFNoC.
- **Stream Endpoint (SEP)**
  The stream endpoint (SEP) provides the high-level flow control for traffic over the network. It also separates control traffic from data traffic to create a separate AXIS-Ctrl network for control traffic. Control traffic refers to things like register reads and writes for configuring and monitoring RFNoC blocks.
- **Control Crossbar**
  The control crossbar is similar to the CHDR Crossbar, but is only for control traffic. It has a much lower throughput and has been optimized for control-traffic, making it much less resource intensive than the CHDR crossbar.
- **Other Blocks**
  The DDC (digital down converter), DUC (digital up converter), and Radio in this example are RFNoC blocks that are included with UHD. These are standard components included the default USRP images and enable typical RF applications. Most blocks only communicate with the RFNoC network, but some blocks require access to external hardware interfaces, such as the radio or DRAM. In this example, the radio blocks connect to the radio hardware on the USRP.

The routes between blocks that go through the crossbar are *dynamic*. That is, the routes can be changed at run time and are controlled by software. The CHDR crossbar is very powerful in that it allows any of its ports to communicate with each other. It allows for streaming between blocks on the same FPGA, between RFNoC blocks and a host computer, or between blocks on different USRPs. New signal processing chains can be added to this crossbar, as needed, for the application.

Similarly, the control crossbar supports dynamic routing, allowing any RFNoC block to send control traffic to any other RFNoC block, even to blocks on a different USRP. Control traffic can also be sent from the host computer to the RFNoC blocks, for example to read/write registers. Additionally, RFNoC blocks can send control to the host computer. In this example, the Radio block sends control traffic to the computer to report specific events, such as overflow or underflow in the radio.

The block connections that don't go through a crossbar (e.g., the connection from the radio to the DDC, and the DDC to the stream endpoint, etc.) are *static* connections. That is, they cannot be changed at run time. Making a connection static has the advantage that it does not require additional ports on the crossbar and that the connection can be made much simpler because the high-level network flow-control protocol used by CHDR is not required. This reduces latency between blocks and leads to FPGA resource savings, allowing more logic be included in a single FPGA image. The static connections are described by the RFNoC image core YAML file. In order to change the static connections, the YAML description needs to be updated and the FPGA image needs to be rebuilt.

The RFNoC framework makes it easy to customize the RFNoC image and add your own IP blocks. Later on in this guide, we'll explore how you can customize the RFNoC image to add or remove blocks, as well as create your own RFNoC blocks and include them in your FPGA builds. This allows you to create highly-customized and high-performance FPGA images for your USRP.

To use RFNoC in UHD 4.0 you need to perform the following:

- Clone the UHD repository
- Install UHD
- Update your device's filesystem and FPGA image

Building custom RFNoC images requires the FPGA source code and build system. These are included in the UHD repository, located in `<repo>/fpga/usrp3/`, where `<repo>` refers to the location where you cloned the UHD repository.

Please see the Binary Installation for UHD installation documentation or Building and Installing instructions to build and install UHD from source. Additionally, AN-445 provides step-by step instructions for cloning the repository and installing UHD from source.

Starting with UHD 3.15, RFNoC is enabled by default, and starting with UHD 4.0, it can no longer be disabled, so no additional instructions are necessary to install support for RFNoC in UHD.

Many of the UHD utilities we will use in this guide are based on Python. Please ensure that your `PYTHONPATH` is set correctly. If not set correctly, you will see errors indicating that Python could not find the `uhd` or `image_builder` libraries. The correct setting may depend on where you installed UHD and which Linux distribution you are using. If you installed UHD from source to the default location on Ubuntu, you may need to add `/usr/local/lib/python3/dist-packages` to your `PYTHONPATH` variable. For example:

```
$ export PYTHONPATH=/usr/local/lib/python3/dist-packages
```

Instructions for updating your device's filesystem (applicable to E3xx and N3xx) and FPGA image can be found in that device's Getting Started Guide. If your filesystem is already up to date, or your device does not use a filesystem (e.g., X3xx), then instructions on flashing an FPGA image to a device can be found in the following locations:

- X3xx series
- E3xx series
- N3xx series

NOTE: FPGA images are specific to the USRP device, NOT the USRP series. For example, a USRP X300 FPGA image will NOT work on a USRP X310 and vice versa. Loading an image that does not correspond to your USRP device will likely lead to an error message but can brick the device under some circumstances.

Make sure you are using the `Bash` shell. Many of the build scripts used by RFNoC are written for `Bash`. See Reconfigure Default Shell in AN-315 for detailed instructions.

Before continuing, please verify that you have downloaded and flashed the latest FPGA image for your version of UHD. All FPGA images for Generation-3 and above devices are RFNoC images, so there is no requirement to download a special image for RFNoC.

Before we start customizing our FPGA, let's get familiar with the default FPGA image, which is pre-built with a set of RFNoC blocks.

Run the following command, with your USRP connected to your PC, to see what is available on the device.

```
$ uhd_usrp_probe --args type=x300
```

Note that your `args` may be different, depending on the USRP device you're using. The example here works for X300/X310. Refer to the manual page on Identifying USRP Devices for more details. If an RFNoC image was successfully loaded onto the USRP, the output will show something like the following:

```
  ___________________________________________________
 /
|       Device: X-Series Device
|     _____________________________________________________
|    /
|   |       Mboard: X310
|   |   revision: 11
|   |   revision_compat: 7
|   |   product: 30818
|   |   mac-addr0: 00:80:2f:17:40:6d
|   |   mac-addr1: 00:80:2f:17:40:6e
|   |   gateway: 192.168.10.1
|   |   ip-addr0: 192.168.10.2
|   |   subnet0: 255.255.255.0
|   |   ip-addr1: 192.168.20.2
|   |   subnet1: 255.255.255.0
|   |   ip-addr2: 192.168.30.2
|   |   subnet2: 255.255.255.0
|   |   ip-addr3: 192.168.40.2
|   |   subnet3: 255.255.255.0
|   |   serial: 311EF81
|   |   FW Version: 6.0
|   |   FPGA Version: 38.0
|   |   FPGA git hash: be53058
|   |
|   |   Time sources:  internal, external, gpsdo
|   |   Clock sources: internal, external, gpsdo
|   |   Sensors: ref_locked
|     _____________________________________________________
|    /
|   |       RFNoC blocks on this device:
|   |
|   |   * 0/DDC#0
|   |   * 0/DDC#1
|   |   * 0/DUC#0
|   |   * 0/DUC#1
|   |   * 0/Radio#0
|   |   * 0/Radio#1
|   |   * 0/Replay#0
|     _____________________________________________________
|    /
|   |       Static connections on this device:
|   |
|   |   * 0/SEP#0:0==>0/DUC#0:0
|   |   * 0/DUC#0:0==>0/Radio#0:0
|   |   * 0/Radio#0:0==>0/DDC#0:0
|   |   * 0/DDC#0:0==>0/SEP#0:0
```

```
|   |   * 0/Radio#0:1==>0/DDC#0:1
|   |   * 0/DDC#0:1==>0/SEP#1:0
|   |   * 0/SEP#2:0==>0/DUC#1:0
|   |   * 0/DUC#1:0==>0/Radio#1:0
|   |   * 0/Radio#1:0==>0/DDC#1:0
|   |   * 0/DDC#1:0==>0/SEP#2:0
|   |   * 0/Radio#1:1==>0/DDC#1:1
|   |   * 0/DDC#1:1==>0/SEP#3:0
|   |   * 0/SEP#4:0==>0/Replay#0:0
|   |   * 0/Replay#0:0==>0/SEP#4:0
|   |   * 0/SEP#5:0==>0/Replay#0:1
|   |   * 0/Replay#0:1==>0/SEP#5:0
...
```

More than this will likely be shown. The specifics of the output depend on the UHD version and which device you're running on. However, we're interested in the following sections:

- **Device description**. At the top, there is a section with basic information about your device, such as serial number, revision, etc. You can use this information to verify that you are indeed communicating with the correct device, and that the device has been updated. The FPGA git hash identifies the commit from which the FPGA image was built.
- **RFNoC blocks on this device**. This section lists all the RFNoC blocks in the loaded FPGA image. The blocks are listed by their block IDs. For example, `0/Radio#0` means that we're on device zero (`0/`), and we're talking about the first radio (with index `#0`). Unless you probe multiple devices at once by specifying multiple addresses in the `--args` argument, the device number will always be zero, but the block index will change depending on the number of blocks of that type. For example, on X310, `Radio#0` corresponds to RF A and `Radio#1` corresponds to RF B.
- **Static connections on this device**. This section lists how the blocks are pre-connected in the FPGA image. For example, the line `0/Radio#0:0==>0/DDC#0:0` means that radio zero, port zero (`:0`), is connected statically to DDC zero, port zero. Static connections can only be changed at compile time when building the FPGA image.

The connection endpoints labeled `SEP` are not RFNoC blocks. They are Stream Endpoints (SEPs). SEPs can send data packets to one another, or to streamers in software, using the CHDR crossbar, which is a dynamic router on the FPGA.

Many RFNoC blocks come with UHD. The HDL source code for these blocks resides in `<repo>/fpga/usrp3/lib/rfnoc/blocks/`. several of the blocks are described below.

Block Name	HDL Name	Description
AddSub	`rfnoc_block_addsub`	Add/Subtract Verilog/VHDL/HLS Example
DmaFIFO	`rfnoc_block_axi_ram_fifo`	FIFO that uses an AXI4 memory-mapped interface for storage. For use with external DRAM or on-chip SRAM.
DDC	`rfnoc_block_ddc`	Digital Down Converter
DUC	`rfnoc_block_duc`	Digital Up Converter
FFT	`rfnoc_block_fft`	Fast Fourier Transform
FIR	`rfnoc_block_fir_filter`	Finite Impulse Response Filter
Fosphor	`rfnoc_block_fosphor`	FFT and waterfall display tool
KeepOneInN	`rfnoc_block_keep_one_in_n`	Keep one sample/packet in N
LogPwr	`rfnoc_block_logpwr`	Computes an estimate of log2(i^2+q^2)
MovingAverage	`rfnoc_block_moving_avg`	Outputs the running average of the N most recent inputs of data stream
NullSrcSink	`rfnoc_block_null_src_sink`	Data source generator, sink, and loopback for testing
Radio	`rfnoc_block_radio`	Radio Interface
Replay	`rfnoc_block_replay`	Record/Playback using AXI4 memory-mapped interface. For use with external DRAM or on-chip SRAM.
SigGen	`rfnoc_block_siggen`	Signal Generator. Supports sinusoidal, constant, and random outputs, with configurable gain.
SplitStream	`rfnoc_block_split_stream`	Splits a single data stream into two
Switchboard	`rfnoc_block_switchboard`	A configurable RFNoC datapath switch for testing
VectorIIR	`rfnoc_block_vector_iir`	Implements an IIR filter with variable length delay line
Window	`rfnoc_block_window`	Windowing module for use with FFT block

In the UHD installation directory, you'll find example applications (e.g., in `/usr/lib/uhd/examples/` or `/usr/local/lib/uhd/examples/`). We'll look at `rfnoc_rx_to_file` to familiarize ourselves with the RFNoC API.

If we look at the source code for this example (located at `<repo>/host/examples/rfnoc_rx_to_file.cpp`), we can see the components necessary in an RFNoC application. In the `UHD_SAFE_MAIN` function, we create a few crucial objects, such as an RFNoC graph, a radio block controller, a DDC block controller, and an RX streamer.

```
auto graph = uhd::rfnoc::rfnoc_graph::make(args);
// ...
auto radio_ctrl = graph->get_block<uhd::rfnoc::radio_control>(radio_ctrl_id);
// ...
uhd::rfnoc::ddc_block_control::sptr ddc_ctrl;
// ...
auto rx_stream = graph->create_rx_streamer(1, stream_args);
```

We also make connections in our graph and configure our blocks.

```
// Connect blocks and commit the graph
for (auto& edge : chain) {
    if (uhd::rfnoc::block_id_t(edge.dst_blockid).match(uhd::rfnoc::NODE_ID_SEP)) {
        graph->connect(edge.src_blockid, edge.src_port, rx_stream, 0);
    } else {
        graph->connect(
            edge.src_blockid, edge.src_port, edge.dst_blockid, edge.dst_port);
    }
}
...
rate = radio_ctrl->set_rate(rate);
...
radio_ctrl->set_rx_frequency(freq, radio_chan);
```

Once our graph and blocks are configured, we use the `recv_to_file` function to receive data from our device and write it to a file on our host computer. Running this example with the `--help` argument shows the available options. For example, to receive 3 seconds of data at a specific frequency and sample rate, we can run:

```
rfnoc_rx_to_file --args type=x300 --freq 2.4e9 --rate 10e6 --duration 3
```

Now you are ready to move to the next step and build your own blocks and FPGA images.

UHD provides tooling to help develop custom RFNoC images. In this guide we will demonstrate how to use the RFNoC Image Builder, which will allow you to change which blocks are included and the connections between blocks. Let's go over common decisions you'll have to make with this tool.

- **Blocks Included**
  The particular blocks to be included in the image is the most impactful decision you'll make in the image building process, both in terms of image functionality and FPGA resource utilization. Adding more blocks means more processing in the image, but it also means longer FPGA build times and less space in the FPGA.
- **Static Connections**
  Recall that blocks can be statically connected, as opposed to connecting every block to the CHDR crossbar (dynamically connected). The number of dynamic connections has a large impact on the size of the CHDR crossbar, so static connections are used to lower resource utilization. However, once a block is statically connected in a chain of blocks, data cannot be sent between arbitrary blocks; it must traverse and be processed by each block in the order defined by the static connections.
- **Hardware Connections**
  All blocks share some connections defined by the RFNoC Specification. These connections are made automatically and cannot be changed. Many blocks require additional connections, such as clocks for the internal DSP or external DRAM interfaces. These connections must also be specified.

Now that we've gone over some of the considerations, let's look at how to actually build a custom RFNoC image. This begins with the RFNoC image core YAML description.

The image YAML file defines RFNoC blocks in the FPGA image. It is designed to be both human-readable and machine-parseable, which allows us to make edits quickly and easily. Each device type has a default YAML image file, which is named `<DEVICE>_rfnoc_image_core.yml` (e.g., `x310_rfnoc_image_core.yml`). Take a look at the RFNoC image core YAML file for the X310 by opening `<repo>/fpga/usrp3/top/x300/x310_rfnoc_image_core.yml`. Notice that it has the following sections.

- **General Parameters**
  Here we define the device, the bit width of our CHDR connections, as well as some versioning and licensing.
- **Stream Endpoints**
  In the `stream_endpoints` section we list each stream endpoint (SEP) that we wish to instantiate. These will be directly connected to the CHDR crossbar and will allow us to make dynamic connections within the CHDR crossbar. Typically you will have one SEP that forms the start and end of a single DSP chain of RFNoC blocks.
  You can also add some per-stream-endpoint configuration here, such as the ingress buffer size, which affects streaming performance from your computer to that SEP. For example, if we know that one SEP will be receiving data transferred from your computer to the USRP then a data buffer is needed to accept those incoming packets, in which case we specify the buffer size by setting the `buff_size` option on that SEP. Alternatively, if we know that a particular SEP only sends data from the USRP to the computer, then we won't need the ingress data buffer and we can set `buff-size` to 0, thus saving FPGA resources.
  Each SEP can have an AXIS-Ctrl and an AXIS-CHDR port, as indicated by the `ctrl` and `data` options. At least one AXIS-Ctrl port is required to communicate with the RFNoC blocks, so `ctrl` typically enabled on just the first SEP. Every SEP will usually have AXIS-CHDR connections to one or more RFNoC blocks, so `data` is usually enabled on all SEPs.
- **NoC Blocks**
  In the `noc_blocks` section we specify all the RFNoC blocks to include in the FPGA image. We'll need to give each block a unique name and reference a block definition YAML file (which we'll discuss in a later section). The blocks chosen have the greatest impact on the functionality of the image, as well as providing very coarse control over the FPGA resource utilization. We'll look at how to change the blocks included in the image as part of our example below.
- **Static Connections**
  The `connections` section defines static connections between blocks, stream endpoints, and various hardware interfaces on the USRP. Statically connected chains of blocks will usually start and end at a stream endpoint (SEP). SEPs are automatically connected to the CHDR crossbar by the RFNoC infrastructure. Remember that data must be passed through the entire statically connected chain; dynamic connections cannot be made to the statically connected blocks in the middle of the chain. Other hardware connections, such as to the external DRAM, would also be specified here. We'll take a closer look at making connections as part of our example below.
- **Clock Domains**
  The `clk_domains` section defines which clocks to connect to each block's clock inputs. Some blocks do not need any clock connections beyond the base clocks required by RFNoC. These required connections are not listed here, since they are always the same for each block. Other blocks may require additional clocks. For example, the radio blocks should be connected to the `radio` clock. Many other blocks require a `ce` (Compute Engine) clock, which is used for the block's internal DSP.

Before we look at editing the image core YAML file, let's go over how to use the RFNoC image builder to generate a bitstream from that YAML file. Additional details on how to run the image builder can be found with the `--help` option:

```
$ rfnoc_image_builder --help
```

Here are some of the options provided:

- **`-y`**: Path to the RFNoC image core YAML file
- **`-t`**: The image target you would like to build. These are the same targets that are used in the FPGA `make` process. More details on these can be found in the Generation 3 USRP Build Documentation of the UHD and USRP Manual. If not specified, the default specified in the image core YAML file will be used.
- **`-l debug`**: Use the `debug` log level. This prints more information about the FPGA connections and available port names, which can be useful for debugging connection errors in the YAML file.
- **`--generate-only`**: Generate the HDL files required, but do not build the FPGA. Users can then build the FPGA later using `make <target>`.
- **`-I`**: Path to the directory containing out-of-tree block YAML descriptions (the YAML files installed with UHD are included by default). This option is only needed if using an out-of-tree RFNoC block.
- **`-F`**: Path to the FPGA source code (e.g., `<repo>/fpga`). This path is only required if the current working directory is not within the UHD repository.

For example, to build the default RFNoC image for X310, you might use the following command:

```
$ cd <repo>/fpga/usrp3/top/x300/
$ rfnoc_image_builder -y ./x310_rfnoc_image_core.yml -t X310_XG
```

In this example `<repo>` refers to the location where you cloned the UHD repository and should be replaced by the location you used. `x310_rfnoc_image_core.yml` is the default RFNoC image core YAML file for the X310, which is in the x300 project directory. `X310_XG` is the make target to use for the build. `XG` in this case refers to dual 10 Gbps Ethernet.

For an out-of-tree RFNoC block, you will also need to specify the location of the block information. For example:

```
$ rfnoc_image_builder -F <repo>/fpga -I <repo>/host/examples/rfnoc-example -y <repo>/host/examples/rfnoc-example/icores/x310_rfnoc_image_c
```

This example shows how to build an FPGA with the Gain example out-of-tree RFNoC block, which is located in `<repo>/host/examples/rfnoc-example/`. The `-F` option is added to specify the location of the FPGA source, and `-I` specifies the location of the out-of-tree block YAML.

The image builder performs several steps in order to build the FPGA image from the image coure YAML:

1. It generates the `<device>_rfnoc_image_core.v` file. This file includes the Verilog code described by the YAML image core file. A `<device>_static_router.hex` file is also generated. This file describes the static connections that should be made by the Verilog code. In the case of the X310 examples above, the output files would be named `x310_rfnoc_image_core.v` and `x310_static_router.hex`, and would be placed in the project directory for the X310 (`<repo>/fpga/usrp3/top/x300/`).
2. The image builder will configure the environment for building the FPGA. This is equivalent to sourcing the `setupenv.sh` script for the device type being built. For example, in our example above, it would run `source <repo>/fpga/usrp3/top/x300/setupenv.sh`.
3. The image builder runs `make <target>` to build the IP and the FPGA bitstream for the indicated target. The generated `rfnoc_image_core.v` and `static_router.hex` are automatically pulled into this build. The completed bitstream will be located in `<repo>/fpga/usrp3/top/{project}/build` directory, which is the same directory created through the normal make process. For example, for X310 it would be located in `<repo>/fpga/usrp3/top/X300/build`.

As an example, let's run through how to modify the YAML file to modify the RFNoC image. Suppose we have an application that we'd like to run on a USRP X310 and we would like to offload the FFT processing to the FPGA. UHD provides an FFT RFNoC block, and the default X310 image has some extra space in it, so we should be able to add this block. First, we copy the default X310 image core file, named `x310_rfnoc_image_core.yml`.

```
$ cd <repo>/fpga/usrp3/top/x300/
$ cp x310_rfnoc_image_core.yml x310_with_fft.yml
```

Now open `x310_with_fft.yml` in your favorite text editor. We'll start by making some room for our new block. The default images use very large ingress FIFO buffers for the main SEPs to maximize streaming performance. But we want to make sure we have enough memory buffer space for our new blocks. So Start by reducing the `buf_size` parameters for `ep0` and `ep2` from 65536 to 32768. If using a device other than X310, the numbers may be different, but you can similarly reduce the `buff_size` parameter by half. On smaller devices, it may be necessary to remove the Replay block to make room for the FFT blocks. After making the changes on X310, the result should look like the following:

```
stream_endpoints:
  ep0:                      # Stream endpoint name
    ctrl: True                    # Endpoint passes control traffic
    data: True                    # Endpoint passes data traffic
    buff_size: 32768              # Ingress buffer size for data
  ...
  ep2:                      # Stream endpoint name
    ctrl: False                   # Endpoint passes control traffic
    data: True                    # Endpoint passes data traffic
    buff_size: 32768              # Ingress buffer size for data
```

Now we can add our FFT block. We'll put it on its own stream endpoint, so we first need to add a new stream endpoint. Add the following to the `stream_endpoints` section:

```
stream_endpoints:
  ...
  ep_fft:               # The name can be incremented from previous SEP
    ctrl: False         # Only the first SEP needs control traffic
    data: True          # We do want to pass data through this SEP
    buff_size: 32768    # Ingress buffer size for data
```

In this example, we've named the SEP `ep_fft`. Other names could be given, as long as the name is unique (it does not have to be numbered). Now that we've allocated an SEP for our block, we need to instantiate the actual block. In the next section, add the following:

```
noc_blocks:
  ...
  fft0:                         # FFT block name
    block_desc: 'fft_1x64.yml'  # Block YAML descriptor file
    parameters:                 # Specify any Verilog module parameters (optional)
      EN_FFT_SHIFT: 1
```

Again, the name `fft0` is arbitrary, but the name must be unique. In some cases, there will also be block parameters you'll want to pass, such as the data format of the FFT output data. In our example, we're setting `EN_FFT_SHIFT` parameter to 1, which causes the FFT block to center the zero frequency bin.

Now that we've added our FFT block, we need to connect it to the stream endpoint. In the next section of the YAML file, we add the static connections:

```
connections:
  ...
  - { srcblk: ep_fft, srcport: out0,  dstblk: fft0,   dstport: in_0 }
  - { srcblk: fft0,   srcport: out_0, dstblk: ep_fft, dstport: in0  }
```

This connects the output of SEP `ep_fft` to the input of block `fft0`, and the output of `fft0` to the input of `ep_fft`. Since we're placing the FFT block on its own SEP, the only connections we need to make are between the SEP and the FFT. All SEPs are automatically connected to the CHDR crossbar, so this effectively connects the FFT block to the crossbar, allowing it to communicate with anything on the RFNoC network.

The names of block ports are defined in the YAML descriptions for the blocks. Blocks can use any names for their ports, and they don't have to be numbered (unless the number of ports is parameterized). Generally, the block ports are named `in_N` for inputs to the block and `out_N` for outputs. SEP ports are named `in0` for the input `out0` for the output.

If you are having trouble connecting a block due to an unresolved connection, running `rfnoc_image_builder` with the `-l debug` option will list all available block ports.

Finally, the FFT block has an additional clock input port named `ce` that is used for the FFT signal processing. We need to connect it to a clock domain. Any clock that's at least as fast as the incoming data rate should be sufficient. For example, X3xx devices have a `ce` clock (214.286 MHz) that is usually a good choice, but `rfnoc_chdr` clock (200 MHz) should also work for this example. For N31x and E3xx devices, `rfnoc_chdr` clock is a good choice (200 MHz on N31x and 100 MHz on E3xx). For N32x, `radio` clock (250 MHz) is a good choice. For example, this is how you would connect the X310's `ce` clock to the `ce` port of the FFT block:

```
clk_domains:
```

```
  ...
  - { srcblk: _device_, srcport: ce, dstblk: fft0, dstport: ce }
```

And this is how you would connect `rfnoc_chdr` clock to the `ce` port:

```
clk_domains:
  ...
  - { srcblk: _device_, srcport: rfnoc_chdr, dstblk: fft0, dstport: ce }
```

And finally, this is how you would connect `radio` clock:

```
clk_domains:
  ...
  - { srcblk: _device_, srcport: radio, dstblk: fft0, dstport: ce }
```

The source block name `_device_` is special and refers to the USRP device itself. Choose a clock that's apprpriate for your device and connect it as shown above.

And that's it! The next step will be to run the image builder on our modified YAML file. Once that's done, and the bitstream has been created, you can load it onto your USRP X310 device, and verify the blocks, as we did in the Inspect the Default Image section.

For example, from the X300 directory, you would run the following command to run the image builder:

```
$ rfnoc_image_builder -y x310_with_fft.yml -t X310_XG
```

To download the FPGA bitstream to the X310, run the following:

```
$ uhd_image_loader --args "type=x300,addr=192.168.30.2" --fpga-path ./build/usrp_x310_fpga_XG.bin
```

You may need to change the IP address to match your device, depending on your configuration. After completing the flash update and power-cycling the USRP, run `uhd_usrp_probe` to confirm that the FFT block was recognized.

```
$ uhd_usrp_probe --args "type=x300,addr=192.168.30.2"
```

Take a look at the RFNoC blocks and the static connections on the device. You should see the following new blocks and connections:

```
|     _____________________________________________________
|    /
|   |       RFNoC blocks on this device:
...
|   |   * 0/FFT#0
...
|     _____________________________________________________
|    /
|   |       Static connections on this device:
...
|   |   * 0/SEP#6:0==>0/FFT#0:0
|   |   * 0/FFT#0:0==>0/SEP#6:0
...
```

Now that we've added the FFT block and verified that it is operating as expected, let's see if we can modify it a little. Let's say that we're sure that we always want to receive the FFT bins from our device, and we don't want to see the raw samples. In order to save some FPGA resources, we can move our FFT block to be between the DDC and the SEP, on the RX data path. Instead of the previous modifications to our YAML file, we want the following:

```
stream_endpoints:
  ...
  # Unchanged from the default image core (no need for ep_fft).

noc_blocks:
  ...
  fft0:                         # FFT block name
    block_desc: 'fft_1x64.yml'  # Block YAML descriptor file
    parameters:                 # Specify any Verilog module parameters (optional)
      EN_FFT_SHIFT: 1

connections:
  ...
  # Change this line:
  # - { srcblk: ddc0, srcport: out_0, dstblk: ep0, dstport: in0  }
  # Change it to the following to add fft0 between ddc0 and ep0:
  - { srcblk: ddc0, srcport: out_0, dstblk: fft0, dstport: in_0 }
  - { srcblk: fft0, srcport: out_0, dstblk: ep0,  dstport: in0 }

clk_domains:
  ...
  # As before, we still connect our FFT block to the clock domain
  - { srcblk: _device_, srcport: rfnoc_chdr, dstblk: fft0, dstport: ce }
```

This last line is valid for E310/E320; for X300/X310/N300/N310/N320/N321 use the following:

```
  - { srcblk: _device_, srcport: ce, dstblk: fft0, dstport: ce }
```

And as simply as that, we have added the FFT block to our RX block chain. Remember that this is a static connection, so when the resulting FPGA image is loaded onto your device, you will no longer be able to receive samples from the DDC directly; all samples on that chain will be processed by the FFT block, and you will only receive FFT bins from this radio chain.

Now that we've gone over what is provided by UHD, let's start to look at custom RFNoC development. It's recommended that custom RFNoC development be kept separate from the in-tree UHD RFNoC infrastructure and examples, in order to simplify version control and licensing. Modules located outside of the repository are called *out-of-tree* (OOT).

In order to understand how OOT modules for RFNoC are created, let's take a look at the example that's provided with UHD. You can find the example located in `<repo>/host/examples/rfnoc-example/`. This example includes a simple Gain RFNoC block that we can use as a reference for creating our own OOT blocks.

Take a look at the `rfnoc-example` directory structure. An OOT module is a collection of block implementations, the software controllers for those blocks, and sometimes applications to demonstrate their functionality. These different components within the OOT module are organized into the directories described below.

- HDL and Image Resources
  - `fpga/`: Contains the HDL (e.g., Verilog) required for the module. Each RFNoC block should have its own subdirectory here, such as `rfnoc_block_gain`, for the Gain block.
  - `blocks/`: Contains the YAML RFNoC block definition files. These describe the block's interfaces and are used by the image builder. We'll go into more detail with regards to what goes into these YAML files in the Creating Your Own RFNoC Block section.
  - `icores/`: Contains example FPGA image core YAML files, to demonstrate how to create an FPGA image using your custom blocks. These image core files may demonstrate important parameters or show recommended configurations for chains of blocks.
- Block Controller and Example Software
  - `include/`: Contains the headers for the block controller software. Once installed, these headers will be used by UHD and other applications to interface with and control your custom RFNoC blocks.
  - `lib/`: Contains the block controller implementations. The files here should implement the features outlined in the headers located in the `include/` directory. All unit test code should also be located here.
  - `apps/`: Contains example applications for your custom blocks. These applications may demonstrate how to use the block controllers, common configurations, or how to process data from your blocks.
- Infrastructure
  - `cmake/`: Contains any custom CMake commands the module may need. In general, simple modules will not need to modify this directory at all. More complicated modules with additional external dependencies may need to modify this.

Take some time to explore the files in the `rfnoc-example`. Note the following files and directories related to the Gain example:

- `rfnoc-example/fpga/rfnoc_block_gain/`
  This is the HDL for the Gain RFNoC block.
  - `rfnoc_block_gain.v`
    The top-level synthesizable file for the Gain block
  - `noc_shell_gain.v`
    The NoC Shell for the Gain block
  - `rfnoc_block_gain_tb.sv`
    The simulation testbench
  - `Makefile`
    The simulation Makefile
- `rfnoc-example/blocks/gain.yml`
  The YAML block definition file for the Gain block
- `rfnoc-example/icores/x310_rfnoc_image_core.yml`
  An example RFNoC image core YAML description showing how to connect the Gain block
- `rfnoc-example/include/rfnoc/example/gain_block_control.hpp`
  The software block controller header
- `rfnoc-example/lib/gain_block_control.cpp`
  The software block controller implementation
- `rfnoc-example/apps/init_gain_block.cpp`
  An example showing how to find and initialize the Gain block

To create your own OOT module, make a copy of the `rfnoc-example` directory. This will contain all the subdirectories and infrastructure files that you need to create your custom module. The directory name for your copy should be renamed to your desired name; we recommend something like `rfnoc-foo`, for example, where `foo` is the name for your OOT module. You'll also need to change `rfnoc-example` within the module's `CMakeLists.txt` files to your module's name.

Let's start by creating our own copy of the `rfnoc-example` to work with. We'll call our module `demo`. You can put the copy wherever you like, but for this example we'll put it in our home directory.

```
$ cp -r <repo>/host/examples/rfnoc-example ~/
$ mv ~/rfnoc-example ~/rfnoc-demo
```

Now we need to edit our `CMakeLists.txt` files to rename our module. Edit the following files and change all instances of `rfnoc-example` to `rfnoc-demo`:

```
~/rfnoc-demo/CMakeLists.txt
~/rfnoc-demo/lib/CMakeLists.txt
```

The OOT module will need to be built and installed in order to use its RFNoC blocks. To build and install the OOT module we just created, do the following:

```
$ mkdir ~/rfnoc-demo/build
$ cd ~/rfnoc-demo/build
$ cmake -DUHD_FPGA_DIR=<repo>/fpga/ ../
```

This configures the project to be installed in the default location. You may want to provide a different install directory to CMake using the `-DCMAKE_INSTALL_PREFIX`. Please refer to the UHD Installation Guide or the installation guide for other CMake-based projects for instructions on configuring CMake.

Note that we specify the location of the FPGA source code using the `-DUHD_FPGA_DIR` option. This allows you to run the FPGA testbenches and to build the example FPGA image provided with the OOT example.

At this point, you can run `make help` to see what options are available. A few of the available make targets are described below:

- `make rfnoc-demo`: Build the block controllers for the RFNoC blocks
- `make testbenches`: Run the testbenches for the RFNoC blocks
- `make x310_rfnoc_image_core`: Build the FPGA example image
- `make install`: Install the module

To build and install the block, perform the following steps.

```
$ cd ~/rfnoc-demo/build
$ make
$ make install
```

Note that `sudo` may be required depending on the install location.

At this point you can build the example FPGA that's included in the `icores` directory. Take a look at `~/rfnoc-demo/icores/x310_rfnoc_image_core.yml`. Note that the `gain` block is added in exactly the same way that the FFT block was in Example: Adding an FFT Block. You can build this example FPGA for the X310 using the following steps:

```
$ cd ~/rfnoc-demo/build
$ make x310_rfnoc_image_core
```

Alternatively, you can call `rfnoc_image_builder` directly. In order to get the image builder to find your custom blocks, you may need to supply some additional arguments to find the OOT module, as shown in the following example:

```
$ rfnoc_image_builder -F <repo>/fpga ?I ~/rfnoc-demo/include/rfnoc -y ~/rfnoc-demo/icores/x310_rfnoc_image_core.yml
```

The `-I` option is only required if the OOT module has not been installed on the system.

If you have an X310, you can build and use the provided example to test the Gain block. If you have a different USRP, you can use its default `rfnoc_image_core.yml` as a starting point and follow the same steps that we followed for the FFT block in Example: Adding an FFT Block, then build it using the process described above.

After building an FPGA image for your USRP and downloading it to your device, the Gain block should be available. Test this by running `uhd_usrp_probe`:

```
$ uhd_usrp_probe --args "type=x300,addr=192.168.30.2"
```

Again, note that your `args` may be different or may not be required if you only have a single USRP connected.

Take a look at the RFNoC blocks and the static connections on the device. You should see the following new blocks and connections:

```
|     _____________________________________________________
|    /
|   |       RFNoC blocks on this device:
...
|   |   * 0/Block#0
...
|     _____________________________________________________
|    /
|   |       Static connections on this device:
...
|   |   * 0/SEP#4:0==>0/Block#0:0
|   |   * 0/Block#0:0==>0/SEP#4:0
...
```

The Gain block is a useful example and could even be used as the starting point for a new block, but rather than trying to copy and edit the Gain block, it is best to use the RFNoC ModTool to create a new block template. This is also required if you want to change the interfaces provided to your IP that are exposed by the NoC Shell.

The first step in creating a new block is to write a YAML description for the block that you want to create. The options available are described RFNoC Specification. You can also look at the blocks available in `<repo>/host/include/uhd/rfnoc/blocks/` for examples. We'll provide a brief overview here.

Open up and take a look at the Gain block definition located in `~/rfnoc-demo/blocks/gain.yml`. Notice the different sections, which are described below:

- **General Parameters**
  In this section we give our block a name (`module_name`) and a unique block ID (`noc_id`). The NoC ID is an arbitrary 32-bit number that will be used by the software to identify the block during system discovery. You can also specify the CHDR bus width, which should be 64 for Generation-3 USRPs.
- **Clocks**
  In the `clocks` section, we define the clocks that will be used by the block. The `rfnoc_chdr` and `rfnoc_ctrl` clocks are required. Many blocks also have a `ce` clock for internal DSP.
- **Control Interface**
  The `control` section defines what type of control interface to make available to your block. The control interface is used for register reads and writes. Most blocks do not need to modify this section and will use the `ctrlport` interface type. Control Port is the standard register interface used by RFNoC. The `clk_domain` parameter allows you to specify which clock domain to use for the register interface exposed to your block. Typically this is the same clock that is used for the `data` interface, to avoid clock crossings.
- **Data Interface**
  The `data` section defines what type of data interface to make available to your block and describes the ports. Most blocks use either the `axis_pyld_ctxt` or the `axis_data` interface types.
  The `clk_domain` parameter allows you to specify which clock domain to use for the data interfaces exposed to your block. Typically this is the same clock that is used for the `control` interface, to avoid clock crossings.
  Under the `inputs` and `outputs` sections you can describe the input and output ports.
- **IO Ports**
  The `io_ports` section is used to describe other device-specific ports to which your block needs to connect (such as the DRAM or radio interfaces). For most blocks, this section is not used.

A few other sections might be present in the YAML file (e.g., `registers` or `properties`). These sections are reserved for future use and can be left empty.

Let's start by making a copy of the Gain block YAML for our use case.

```
$ cp ~/rfnoc-demo/blocks/gain.yml ~/rfnoc-demo/blocks/demo.yml
```

Now open the `demo.yml` file you just created and make the following changes:

1. Change the name of the block from `gain` to `demo`
2. Change the `noc_id` to `0x0000DE30`
3. Change the `format` for the `in` and `out` ports from `int32` to `sc16`.

This YAML file now describes a block with the following features:

- The block named "demo"
- NoC ID of 0xDE30
- CtrlPort register interface on the `rfnoc_chdr` clock domain
- A single 32-bit input port named "In" on the `rfnoc_chdr` clock domain that will provide sc16 samples to your IP

- A single 32-bit output port named "Out" on the `rfnoc_chdr` clock domain that will receive sc16 samples from your IP

Now that we have a block definition YAML file, we can generate the source code templates and NoC Shell for our block. To do so, run the following command:

```
$ python3 <repo>/host/utils/rfnoc_blocktool/rfnoc_create_verilog.py -c ~/rfnoc-demo/blocks/demo.yml -d ~/rfnoc-demo/fpga/rfnoc_block_demo
```

This will create a folder named `~/rfnoc-demo/fpga/rfnoc_block_demo` with an RFNoC block template for you to use. Explore the folder that was created. You should see the following files:

- **`Makefile`**: This is the Makefile for the simulation testbench. See Running a Testbench in the UHD and USRP Manual for instructions on how to run a testbench.
- **`Makefile.srcs`**: This is another makefile that identifies the HDL source code that makes up your RFNoC block.
- **`noc_shell_demo.v`**: This is the NoC Shell that was generated for your block. It provides the interfaces described in the block definition YAML.
- **`rfnoc_block_demo.v`**: This is a template for your RFNoC block. It includes all the top-level ports required by RFNoC and instantiates the NoC Shell.
- **`rfnoc_block_demo_tb.sv`**: This is a testbench template for your RFNoC block. It instantiates your RFNoC block and the bus functional models (BFMs) needed to communicate with your block in simulation.

Take a moment to look at the `rfnoc_block_demo.v` and `rfnoc_block_demo_tb.sv` files that were generated for your block. Notice the "User Logic" section towards the end of the `rfnoc_block_demo` module. This is the location where you can insert your own IP.

The input data samples come in on the `m_in_payload` AXI4-Stream bus. The output data samples go out on the `s_out_payload` AXI4-Stream data bus.

If your input packets are the same size as your output packets (i.e., for every sample in you generate one sample out) then the `m_in_context` data bus can be used to drive `s_out_context`. The context bus is described in detail in the RFNoC Specification.

Similarly, the testbench has a section where you can add your own test sequences to the test bench.

If you decide you need to change the RFNoC interfaces for your block (e.g., add ports, or change the interface type), then it is recommended to update the block definition YAML then regenerate the block. Please take care to not overwrite any code you have written when regenerating the files for your block! Rerunning the tool will cause new files to be generated, with a new NoC Shell, and will demonstrate how to update your code for the new interfaces.

Refer to the Gain example and other RFNoC blocks for examples of how to complete your RFNoC block logic and testbenches.

In this guide we have given a brief introduction on how to develop for RFNoC in UHD 4.0. There's a lot more you can do than is described here, but this will hopefully get you started. Happy coding!